IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Applicant: Hershkovich et al.	§	
Serial No.: 10/690,556	§	
Filed: October 23, 2003	§	Group Art Unit: 2162
For: Search Method Using Coded Keys	§	Attorney Docket: 2694/24
Examiner: **Jean B. Fleurantin**	§	

Commissioner of Patents and Trademarks
Alexandria, Virginia 22313

FAX NO. 571-273-4035 (10 pages)

INFORMAL

Examiner Fleurantin:

Further to our conversation, I have written out – in brief -- some of the main issues for discussion. After discussing these issues, we may want to explore with you some subject matter for additional claims.

Looking forward to speaking with you Wednesday, 5 December, at 3PM EST.

Yours,

Mark Friedman

1) Examiner's Action did not address Applicant's explicit assertion (in the last Office Action Response) that Stark and Fuh are not properly combinable:

> Applicant further argued that Stark '446 and Fuh '662 are not properly combinable. Although, as the Examiner has written, Fuh claims that the transformation module "improves data integrity" [column 8, lines 36-40], Applicant respectfully articulated that the transformation module of Fuh '662 could not improve Stark's data integrity. While Fuh's claim of improved data integrity may be relevant with respect to systems containing "user-defined data" [Fuh '662 column 2, lines 20-25], it is manifest that Stark '446 has no problem of data integrity that Fuh's teachings can solve.

Applicant continues to steadfastly maintain that Stark and Fuh are not properly combinable.

Please note that Fuh's teachings explicitly and specifically relate to relational database systems. Like in a "Google" search, a poor choice or representation of the input keys can result in poor "data integrity", such that the results are not what the user intended to retrieve.

By sharp contrast, Stark teaches databases having a one-to-one correspondence between input key and associated data. This is more like looking for an entry in a dictionary. There is, at most, a singular, unique key that matches the input key. Also, once a match is found, there is a unique field of associated data associated with the entry. There is no "guessing" like in Fuh.

In computer science terms, a dictionary is a dynamic set, the set having unique key values (i.e., all the keys are different). Hereinbelow I have included the computer science definition of "dictionary" from the NIST (National Institute of Standards and Technology), available at http://www.nist.gov/dads/HTML/dictionary.html.

I have further included two pages from Thomas H. Coren et al.: Introduction to Algorithms, MIT Press, a classic and prestigious reference on algorithms, in which the terms "dictionary" and "dynamic set" are defined.

It is manifest that neither Fuh nor Kimura fulfill the basic requirement of "dictionaries", and their teachings are irrelevant and not applicable to the teachings of Stark, whose method manifestly pertains solely to dynamic sets having unique satellite (associated) data.

There is no problem with data integrity in dictionaries, because of the one-to-one correspondence described above. Google-type key searches are not relevant for dictionaries. Consequently, there is no genuine motivation for combining the art of Fuh with the art of Stark.

2) Examiner's Action combines Stark and Fuh and Kimura. Since Stark and Fuh are not properly combinable, the additional combination of Kimura is improper. Moreover, above and beyond this point, the Examiner's cited motivation for combining Kimura with Stark and Fuh appears incomprehensible to us: "to provide number of bits that are included in the offset field depends upon the location of the pattern of data in the sequence of data". This appears to have nothing to do with Stark, nothing to do with Fuh, and nothing to do with whatever very narrow common denominator Stark + Fuh have.

For example: Stark has no offset fields whatsoever. Since Stark is monotonically ordered, an offset field is demonstrably wasteful and absolutely unnecessary. Kimura's art only works with pointers. Stark's art is pointer-less, and this is actually one of the great advantages of Stark. Thus, one skilled in the art would

not neutralize the inherent advantages, economies and efficiencies of Stark by trying to introduce the pointer-based technology of Kimura.

Thus, for all these reasons, Kimura is not properly combinable with Stark and Fuh.

3) Examiner maintains Kimura teaches the log base 2 transformation of key entries disclosed by the instant invention. Applicant argues that a close reading of Kimura shows that Kimura does not teach such a transformation of key entries.

We begin with a short explanation of the Kimura method:

Kimura relates to "on the fly" file data compression and decompression. The purpose is efficient file storage on secondary storage (a disk) of a computer.

It is based upon the following assumptions:

- A data file is divided into autonomous units called chunks (as in the classical LZRW method). While the entire file is held in the secondary storage, the chunk (compressed and decompressed is held in the main memory.
- Compression and decompression is performed on chunks. A chunk is compressed and decompressed independently of other file chunks.
- The compression/decompression method is based upon the compressed/decompressed chunk being located in the main computer memory. This memory is randomly accessed (i.e., chunk blocks can be accessed in any order or sequence). This facilitates a single-pass compression/decompression of the chunk (in contrast to disks, in which data can be accessed in a sequential fashion only. Therefore, single-pass compression/decompression is not feasible, since compression /decompression of data would require the retrieval of the entire file).
- The compression /decompression method of Kimura is an improvement to the LZRW1 method. The LZRW compression method is based on the premise that many blocks of data in the chunk are repetitive. During compression, if a particular block of the chunk's data has been identified as being already compressed, rather than repeating the compression once again and storing once again the compressed data, one can store the previous location and the block length which has been already compressed. This location is indicated as an offset (expressed in number of bytes in relationship to the prior occurrence of the block) in relationship to the current position within the chunk, and the block's length (expressed also in number of bytes). Since the main memory can be randomly accessed, such a block is instantaneously accessed, thus facilitating on the fly compression/decompression. If the blocks are large (hundreds, or thousands of bytes), while the offset and the block length consume a few bytes, this method represents a substantial savings in storage. As rather than repeating the entire block (compressed or uncompressed), the program can access immediately (random access) the already compressed instance of the same data. This represents a rather substantial savings in storage.
- Kimura's method is a further improvement to the basic LZRW1 algorithm in efficient representation of the offset and length fields of a previously occurred instance of this

block. This is stated in Kimura column 2, lines 60-65: "The present invention recognizes that all of the bits allocated to the offset field by LZRW1 are not always needed and as such allocating all of the bits may constitute a waste of memory space. The present invention eliminates this waste by using an approach that uses variable-sized length and offset fields in copy tokens." (see also the table in column 6 of Kimura)

Note: The "copy token" is the position and size of the recently handled block instance, which takes the place of the block data itself.

Kimura is based on offset fields indicating the location of particular data, while both Stark and the instant invention relate to ordered databases. Stark is monotonically ordered, hence, an offset field is clearly wasteful and absolutely unnecessary.

Explanation of lines 32-45 in column 7
How the Kimura's Copy Token eliminates the "waste" and provides the claimed savings:

The entire method is based upon keeping the Copy Token fixed in size. This is demonstrated in Fig. 11B which shows a Copy Token of 16 bits, i.e.; fixed in length. This is the improvement over the original LZRW1 algorithm. This is accomplished by the Offset and the Length fields varying in length, but alwaus summing-up to 16 bits.

Therefore, the Kimura's method can handle various combinations of Offset/Length pairs, as demonstrated in the table of column 6, lines 15-25. He calls these combinations a "sliding window".

The lines 32-45 in column 7 describe hoe an Offset/Length pair is selected from one of the options which appears in the table. The process is fully described by the flowchart in FIG. 13. This flowchart basically says:

1. If the block which instance has already occurred is two bytes or less, don't bother to replace it by the Copy Token, since the Copy Token is also 2 bytes (16 bits) and no storage savings will be accomplished. In this case the 1, or two byte data is left as-is (literal).
2. If however a match is found to be longer than two bytes, the Copy Token selects the longest possible match. Here the longest possible match is in relationship to the Offset/length combinations, which options are listed in the table of column 6, lines 15-25.

FIG. 14 is a flowchart which details the method by which the longest match and eventually an Offset/Length pair are selected. It uses the following recipe:

1. First calculate the offset. This is the difference in between the current pointer position and the location of the previous block's instance. This distance based upon the column 6, page table can assume one of the ranges in the table.

Example: The distance (also called a displacement range) is 751. This corresponds to the 7th displacement range option in the table of column 6, lines 15-25. To represent any number in this displacement range requires 10 bits.

The number of offset bits required is given by:

$$N^{o}\_of\_Offset\_bits = \log_2 (CP - SB) \qquad (I)$$

wherein:
CP – Current Position Pointer
SB- Start Buffer Pointer, which is the previous location occurrence of the same block.

2. Once the number of bits required for offset is determined, the number of Copy Token bits left for the block's length is given by:

$$N°\_of\_Length\_bits = 16 - N°\_of\_Offset\text{-}bits \qquad (II)$$

It is manifest that formula (I) of Kimura relates solely to reserving space for pointing to where the data is located, and has nothing to do with the actual content of the key. This in no way resembles the instant invention in which a key entry is transformed into a coded key that retains a significant portion of the original data, yet has an extremely short length in relation to the full key entry, i.e., substantially equal to the log-base 2 of the full key entry.

4) Examiner's 112 rejection appears unfounded, and comes after previous rounds of examination failed to find any 112 problem in claims 21-39.

5) Examiner's double-patenting rejection appears unfounded. Stark does not "transform" keys. He simply has an additional 0 or 1 bit indicating an open or closed range boundary. This has nothing to do with the actual, substantial content of the key.

<u>other patentable features for discussion:</u>

6) uni-directional transformation

Kimura performs compression and decompression and is therefore bi-directional.

7) unique matching

8) search coded keys, then search full keys

Implementation

(C++, Pascal, and Fortran). Kaz Kylheku's Kazlib (C) implementing dictionary, dynamic hash table, red-black tree, and doubly linked list. Herbert Glarner's Patricia tree (Linoleum) implementation.

Go to the Dictionary of Algorithms and Data Structures home page.

If you have suggestions, corrections, or comments, please get in touch with Paul E. Black.

Entry modified 21 May 2007.
HTML page formatted Mon May 21 08:45:03 2007.

Cite this as:
Paul E. Black, "dictionary", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. 21 May 2007. (accessed TODAY)
Available from: http://www.nist.gov/dads/HTML/dictionary.html

Some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the real numbers, or the set of all words under the usual alphabetic ordering. (A totally ordered set satisfies the trichotomy property, defined on page 31.) A total ordering allows us to define the minimum element of the set, for example, or speak of the next element larger than a given element in a set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: ***queries***, which simply return information about the set, and ***modifying operations***, which change the set. Here is a list of typical operations. Any specific application will usually require only a few of these to be implemented.

SEARCH(S, k) A query that, given a set S and a key value k, returns a pointer x to an element in S such that $key[x] = k$, or NIL if no such element belongs to S.

INSERT(S, x) A modifying operation that augments the set S with the element pointed to by x. We usually assume that any fields in element x needed by the set implementation have already been initialized.

DELETE(S, x) A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation uses a pointer to an element x, not a key value.)

MINIMUM(S) A query on a totally ordered set S that returns the element of S with the smallest key.

MAXIMUM(S) A query on a totally ordered set S that returns the element of S with the largest key.

SUCCESSOR(S, x) A query that, given an element x whose key is from a totally ordered set S, returns the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x) A query that, given an element x whose key is from a totally ordered set S, returns the next smaller element in S, or NIL if x is the minimum element.

The queries SUCCESSOR and PREDECESSOR are often extended to sets with nondistinct keys. For a set on n keys, the normal presumption is that a call to MINIMUM followed by $n - 1$ calls to SUCCESSOR enumerates the elements in the set in sorted order.

The time taken to execute a set operation is usually measured in terms of the size of the set given as one of its arguments. For example, Chapter 14 describes a data structure that can support any of the operations listed above on a set of size n in time $O(\lg n)$.

Introduction

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets *dynamic*. The next five chapters present some basic techniques for representing finite dynamic sets and manipulating them on a computer.

Algorithms may require several different types of operations to be performed on sets. For example, many algorithms need only the ability to insert elements into, delete elements from, and test membership in a set. A dynamic set that supports these operations is called a *dictionary*. Other algorithms require more complicated operations. For example, priority queues, which were introduced in Chapter 7 in the context of the heap data structure, support the operations of inserting an element into and extracting the smallest element from a set. Not surprisingly, the best way to implement a dynamic set depends upon the operations that must be supported.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose fields can be examined and manipulated if we have a pointer to the object. (Chapter 11 discusses the implementation of objects and pointers in programming environments that do not contain them as basic data types.) Some kinds of dynamic sets assume that one of the object's fields is an identifying *key* field. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may contain *satellite data*, which are carried around in other object fields but are otherwise unused by the set implementation. It may also have fields that are manipulated by the set operations; these fields may contain data or pointers to other objects in the set.

http://www.nist.gov/dads/HTML/dictionary.html

NIST
National Institute of Standards and Technology

dictionary

(data structure)

Definition: An *abstract data type* storing items, or values. A value is accessed by an associated *key*. Basic operations are new, insert, find and delete.

Formal Definition: The operations new(), insert(k, v, D), and find(k, D) may be defined with *axiomatic semantics* as follows.

1. new() returns a dictionary
2. find(k, insert(k, v, D)) = v
3. find(k, insert(j, v, D)) = find(k, D) if $k \neq j$

where k and j are keys, v is a value, and D is a dictionary.

The modifier function delete(k, D) may be defined as follows.

4. delete(k, new()) = new()
5. delete(k, insert(k, v, D)) = delete(k, D)
6. delete(k, insert(j, v, D)) = insert(j, v, delete(k, D)) if $k \neq j$

If we want find to be a *total function*, we could define find(k, new()) using a special value: *fail*. This only changes the return type of find.

7. find(k, new()) = *fail*

Also known as association list, map, property list.

Generalization (I am a kind of ...)
binary relation, *abstract data type*.

Specialization (... is a kind of me.)
associative array.

See also *total order*, *set* Some implementations: *linked list*, *hash table*, *B-tree*, *jump list*, *directed acyclic word graph*.

Note: The terms "association list" and "property list" are used with LISP-like languages and in the area of Artificial Intelligence. These suggest a relatively small number of items, whereas a dictionary may be quite large. Professionals in the Data Management area have specialized semantics for "dictionary" and related terms.

A dictionary defines a binary relation that maps keys to values. The keys of a dictionary are a set.

Contributions by Rob Stewart 16 March 2004.

Author: PEB